CHALMERS
UNIVERSITY OF TECHNOLOGY

Bytecode-Based Multiple Condition Coverage: An Initial Investigation

Bytecode-based Multiple Condition Coverage: An Initial Investigation

Srujana Bollina[1] and Gregory Gay[2]

[1] University of South Carolina, Columbia, SC, USA, sbollina@email.sc.edu
[2] Chalmers and the University of Gothenburg, Gothenburg, SE, greg@greggay.com

Abstract. Masking occurs when one condition prevents another from influencing the output of a Boolean expression. Adequacy criteria such as Multiple Condition Coverage (MCC) overcome masking within one expression, but offer no guarantees about subsequent expressions. As a result, a Boolean expression written as a single complex statement will yield more effective test cases than when written as a series of simple expressions. Many approaches to automated test case generation for Java operate not on the source code, but on bytecode. The transformation to bytecode simplifies complex expressions into multiple expressions, introducing masking. We propose Bytecode-MCC, a new adequacy criterion designed to group bytecode expressions and reformulate them into complex expressions. Bytecode-MCC should produce test obligations that are more likely to reveal faults in program logic than tests covering the simplified bytecode.
A preliminary study shows potential improvements from attaining Bytecode-MCC coverage. However, Bytecode-MCC is difficult to optimize, and means of increasing coverage are needed before the technique can make a difference in practice. We propose potential methods to improve coverage.

Keywords: Search-Based Test Generation, Adequacy Criteria, Coverage Criteria

1 Introduction

For any reasonably complex software project, testing alone cannot prove the absence of faults. As we cannot know what faults exist a priori, dozens of *adequacy criteria*—ranging from the measurement of structural coverage to the detection of synthetic faults [8]—have been proposed to judge testing efforts. In theory, if the goals set forth by such criteria are fulfilled, tests should be *adequate* at detecting faults related to the focus of that criterion. Adequacy criteria such as Statement or Branch Coverage have proven popular in both research and practice, as they are easy to measure, offer clear guidance to developers, and present an indicator of progress [9]. Adequacy criteria also play an important role in search-based test generation, as they offer optimization targets that shape the resulting test suite [16].

Masking occurs when one condition—an atomic Boolean variable or subexpression—prevents another condition from influencing the output of the expression. Even if a fault in a Boolean expression is triggered, other parts of that expression—or future expressions encountered along the path of execution—can prevent that fault from triggering an observable failure during test execution.

Sophisticated logic-based adequacy criteria such as Multiple Condition Coverage (MCC) or Multiple Condition/Decision Coverage (MC/DC) are designed to overcome

masking within a single expression. However, they can offer no guarantees about masking in subsequent expressions. As a result, such criteria are sensitive to how expressions are written [8]. A Boolean expression written as a single complex statement will more effective test cases than the same expression written as multiple simple expressions, as the adequacy criterion will not prevent masking between expressions.

Many approaches to automated analysis and test case generation for Java operate not on the source code, but on the resulting bytecode [4,19]. The transformation from source to bytecode translates complex expressions into multiple simple expressions, introducing the risk of masking between expressions. This could limit the fault-finding potential of bytecode-based adequacy criteria. To overcome this limitation, we propose a new variant of Multiple Condition Coverage.

Our approach, **Bytecode-MCC**, is a new test coverage criteria that prescribes a set of test obligations—goals that must be satisfied by test cases—for a class-under-test. Bytecode-MCC groups related Boolean expressions from the bytecode, reformulates the grouping into a single complex expression, and calculates all possible combinations of conditions within the constructed expression. Bytecode-MCC should produce test obligations that—when satisfied—are more likely to reveal faults in the program logic than tests providing simple coverage over the simplified bytecode.

Bytecode-MCC can be used to measure the power of existing test suites or as a target for automated test generation. To examine both scenarios, we have implemented an algorithm to generate test obligations and measure coverage in the EvoSuite search-based test generation framework [4]. We have also implemented a fitness function within EvoSuite intended to enable the automated generation of test suites.

We conducted a preliminary study examining the effectiveness of test generation targeting Bytecode-MCC on 109 faults from Defects4J—a database of real faults from Java projects [10]. Results attained for the "Time" system, where targeting the combination of Bytecode-MCC and Branch Coverage yields an average of 92% Bytecode-MCC coverage, yield an average 32.50%-35.00% likelihood of fault detection—well over the overall average. This suggests the potential of approaches that can attain high Bytecode-MCC coverage.

However, the results for other systems are more negative. Bytecode-MCC is difficult to optimize, and our fitness function does not offer sufficient feedback to guide test generation. Additional search budget does not guarantee higher levels of coverage. This suggests that Bytecode-MCC may be best used as a method of judging test suite quality, rather than as a direct generation target. Simultaneously targeting Bytecode-MCC and Branch Coverage improves coverage of Bytecode-MCC and improves the likelihood of fault detection, as the fitness function for Branch Coverage offers more feedback to the search process. Therefore, other fitness functions may offer the means to satisfy Bytecode-MCC. Bytecode-MCC has potential to yield effective test suites if the identified limitations can be overcome. We propose suggestions on how to proceed in future work and make our implementation available.

2 Background

Adequacy Criteria: Adequacy criteria are important in providing developers with the guidance they need to test efficiently, as they identify inadequacies in the test suite. For

example, if a given test does not reach and execute a statement, it is inadequate for finding faults in that statement.

Each adequacy criterion prescribes a series of *test obligations*—goals that must be met for testing to be considered "adequate" with respect to that criterion. Often, such criteria are structured around particular program elements and faults associated with those elements, such as statements, branches of control flow, or Boolean conditions [8]. When a criterion has been satisfied, the system is considered to be adequately tested with respect to that element. Adequacy criteria have seen widespread use, as they offer objective, measurable checklists [9].

In this study, we are concerned with adequacy criteria defined over Boolean *decisions*, complete Boolean expressions within a program. Decisions can be broken into simple *conditions*—atomic Boolean variables or subexpressions—connected with operators such as `and`, `or`, `xor`, and `not`.

- **Decision Coverage:** This simple criterion requires that all decision statements evaluate to both possible outcomes—`true` and `false`. Given the expression $(A \textit{ or } B)$, the test suite `(TT),(FF)` attains decision coverage over that expression.
- **Branch Coverage:** The source code of a program can be broken into basic blocks—sets of statements executed sequentially. Branches are decision statements that can decide which basic blocks are executed, such as `if`, `loop`, and `switch` statements. Branch Coverage requires that the test suite cover each outcome of all branches. Improving branch coverage is a common objective in test generation [13].
- **Multiple Condition Coverage (MCC):** MCC requires test cases that guarantee all possible combinations of condition outcomes within the decision to be executed at least once. Given expression $(A \textit{ or } B)$, MCC coverage requires the value combinations `(TF),(TT),(FF),(FT)`. MCC is more expensive to attain than Decision Coverage, but offers greater potential fault-detection capability. Note that, in the presence of short-circuit evaluation, infeasible outcomes are not required. In the previous example, short-circuit evaluation would reduce the required test suite to `(FF),(FT),(T-)`.

Search-Based Software Test Generation: Selection of test inputs is generally a costly manual task. However, given a measurable testing goal, input selection can be framed as a *search* for the input that achieves that goal. Automation of input selection can potentially reduce human effort and the time required for testing [13].

Meta-heuristic search provides a possible solution for test input generation. Given scoring functions denoting *closeness to the attainment of those goals*—called *fitness functions*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*). Metaheuristics are often inspired by natural phenomena. For example, genetic algorithms evolve a group of candidate solutions by filtering out bad "genes" and promoting fit solutions [4]. Due to the non-linear nature of software, resulting from branching control structures, the search space of a real-world program is large and complex. Metaheuristic search—by strategically sampling from that space—can scale effectively to large problems. Such approaches have been applied to a wide variety of testing scenarios [2]. Adequacy criteria are ideal as test generation targets, as such criteria can be straightforwardly translated into the fitness functions used to guide the search [16].

3 Bytecode-Based Multiple Condition Coverage

Masking occurs when a condition, within a decision statement, has no effect on the value of the decision as a whole. As an example, consider the trivial program fragments to the right. The program fragments have different structures, but are functionally equivalent.

Version 1: Complex Implementation

```
out_1 = (in_1 or in_2) and in_3;
```

Version 2: Simple Implementation

```
expr_1 = in_1 or in_2;
out_1 = expr_1 and in_3;
```

Version 1 presents the full, complex expression. Version 2 is defined using intermediate variable `expr_1`. Given a decision of the form `in_1 or in_2`, the truth value of `in_1` is irrelevant if `in_2` is true, so we state that `in_1` is *masked out*. Masking can have negative consequences on the testing process by preventing the effect of a fault from propagating to a visible failure.

MCC is able to overcome masking within a single expression by requiring that all possible combinations of condition values be attempted, meaning that non-masking test cases must exist. However, MCC is sensitive to how expressions are written. Variable `in_3` can have a masking effect—when it is `false`, it determines the value of the decision it is in. In the complex implementation, MCC would require test cases that overcome this masking effect, showing the effect that `in_1` and `in_2` have on the overall decision. In the simple, multi-line case, we only require that `in_3` be evaluated with the overall expression `expr_1`.

Suppose this code fragment is faulty and the correct expression should have been `in_1` **`and`** `in_2`. Tests over the simplified implementation may miss this fault, while any test set providing coverage of the complex implementation would reveal this fault. This can have significant ramifications with respect to fault finding of test suites [8,7,18]. The simplified version can be more trivially satisfied, with fewer test cases, than cases where the code is structured into fewer, more complex expressions. The complex version will have more complex test obligations and will generally require more test cases, but those test cases will generally have more fault revealing power [8].

Many approaches to automated analysis and test case generation for Java operate not on the source code, but on the bytecode [4,19]. Bytecode is often easier to instrument than the source code—for instance, it can be obtained without the source code being present—and bytecode-based techniques are often more efficient and scalable than source code-based techniques [19]. Many state-of-the-art techniques compute coverage and generate test cases by monitoring the instrumented bytecode [4].

The transformation from source to bytecode requires a similar simplification. Consider the example depicted in Figure 1, where the source code is shown on the left and the bytecode is shown on the right. The complex statement on the left is translated into a series of simple expressions. As a result of this transformation, the risk of masking is introduced between expressions. As all expressions are maximally simplified, a straight-forward implementation of MCC would be equivalent to Branch Coverage over each individual statement.

Given concerns over the fault-revealing power of tests generated over simplified representations of Boolean expressions [8,7]—as well as concerns over whether code coverage attained over bytecode accurately predicts coverage over the source code [12]—

```
package tutorial;

class Sample
{
    public void myMethod(int a, int b, int c, boolean status) {

        if (status && (a != 0 || b > c || a != b ) ) {
                // true Case
            if (c <= a) {
                // continue here...
                    } else {
                        //return "
                    }
        } else {
            // false case
        }
    }

}
```

```
1 Method Name: <init>()V
2 I0 (0) LABEL L554976255
3 I1 (0) LINE  4
4 I2 (0) ALOAD 0
5 I3 (1) INVOKESPECIAL java/lang/Object.<init>()V
6 I4 (4) ALOAD 0
7 I5 (5) INVOKESTATIC org/evosuite/runtime/System.registerOb
8 I6 (8) RETURN
9 I7 (9) LABEL L411754329
10 Method Name: myMethod(IIIZ)V
11 I0 (0) LABEL L2061041761
12 I1 (0) LINE  8
13 I2 (0) ILOAD 4
14 I3 (2) UNKNOWN Branch I3 IFEQ, jump to L1862090560
15 I4 (5) ILOAD 1
16 I5 (6) UNKNOWN Branch I5 IFNE, jump to L550284715
17 I6 (9) ILOAD 2
18 I7 (10) ILOAD 3
19 I8 (11) UNKNOWN Branch I8 IF_ICMPGT, jump to L550284715
20 I9 (14) ILOAD 1
21 I10 (15) ILOAD 2
22 I11 (16) UNKNOWN Branch I11 IF_ICMPEQ, jump to L1862090560
23 I12 (19) LABEL L550284715
24 I13 (19) LINE  10
25 I14 (19) ILOAD 3
26 I15 (20) ILOAD 1
27 I16 (21) UNKNOWN Branch I16 IF_ICMPGT, jump to L1862090560
28 I17 (24) LABEL L1862090560
29 I18 (24) LINE  18
30 I19 (24) RETURN
31 I20 (25) LABEL L160656677
```

Fig. 1: A complex Boolean expression. Source code is shown on the left, and its equivalent bytecode is shown on the right.

test generation approaches able to account for this simplification may yield more effective and representative testing results.

To overcome the limitations imposed by the translation to a simplified program structure, we propose a new variant of MCC for bytecode, **Bytecode-MCC**. Bytecode-MCC groups related Boolean expressions, reformulates the grouping into a single complex expression, and calculates all possible combinations of conditions within the constructed expression. Bytecode-MCC should produce test obligations that—when satisfied—are likely to reveal faults in program logic.

Bytecode-MCC can be used to assess existing test suites as well as as a target for automated test generation. To examine both scenarios, we have implemented an algorithm to generate test obligations and measure coverage in the EvoSuite test generation framework [4]. We have also implemented a fitness function intended to enable the automated creation of Bytecode-MCC-satisfying test suites.

Our implementation of Bytecode-MCC as a fitness function and coverage measurement mechanism in EvoSuite is available from `https://github.com/Srujanab09/evosuite`.

3.1 Test Obligation Generation

To formulate the test obligations for Bytecode-MCC, we perform the following process:

1. Search the bytecode for Boolean expressions.
2. When an expression is detected, begin building a group of related expressions.
3. Add any subsequent Boolean expressions in the same bytecode label—a basic block of sequentially executed expressions—to the grouping.
4. When a new label is reached, add any new Boolean expressions to that grouping.

```java
package tutorial;

class Sample
{

    public void myMethod(int a, int b, int c, int d) {
        if (a > 555) {
            if (b < 9) {
                if ( c >= 788) {
                    if (d != 909) {
                // continue here...
                    } else {
                        //return "D failed"
                    }
                } else {
                    //return "C failed";
                }
            } else {
                //return "B failed";
            }
        } else {
            //return "A failed";
        }

    }


}
```

```
Method Name: <init>()V
I0 (0) LABEL L554976255
I1 (0) LINE  4
I2 (0) ALOAD 0
I3 (1) INVOKESPECIAL java/lang/Object.<init>()V
I4 (4) ALOAD 0
I5 (5) INVOKESTATIC org/evosuite/runtime/System.registerOb
I6 (8) RETURN
I7 (9) LABEL L411754329
Method Name: myMethod(IIII)V
I0 (0) LABEL L1899798671
I1 (0) LINE  8
I2 (0) ILOAD 1
I3 (1) INT 555 Type=null, Opcode=SIPUSH
I4 (4) UNKNOWN Branch I4 IF_ICMPLE, jump to L1871531303
I5 (7) LABEL L717117575
I6 (7) LINE  9
I7 (7) ILOAD 2
I8 (8) INT 9 Type=null, Opcode=BIPUSH
I9 (10) UNKNOWN Branch I9 IF_ICMPGE, jump to L1871531303
I10 (13) LABEL L1866234461
I11 (13) LINE  10
I12 (13) ILOAD 3
I13 (14) INT 788 Type=null, Opcode=SIPUSH
I14 (17) UNKNOWN Branch I14 IF_ICMPLT, jump to L1871531303
I15 (20) LABEL L2064685037
I16 (20) LINE  11
I17 (20) ILOAD 4
I18 (22) INT 909 Type=null, Opcode=SIPUSH
I19 (25) UNKNOWN Branch I19 IF_ICMPEQ, jump to L1871531303
I20 (28) LABEL L1871531303
I21 (28) LINE  26
I22 (28) RETURN
I23 (29) LABEL L2121337309
```

Fig. 2: A simple Java class. Source code is on the left and bytecode is on the right.

5. Stop when a label is reached with no Boolean expressions.
6. Formulate a truth table containing all evaluations of the gathered expressions.
7. Translate each row of the truth table into a test obligation.

For a given class and method, we inspect the bytecode to gather related Boolean expressions. In the bytecode, expressions are grouped into labels. A label indicates the start of a series of sequentially-executed expressions, and is a point that another control-altering expression can jump to. While monitoring the bytecode, we start a grouping when we detect a Boolean expression. Each Boolean expression in bytecode is represented using a form of `if-statement` where a `true` outcome causes a jump to another label. We add this *if-statement* to our grouping, noting the label that is jumped to if the statement evaluates to `true` and where we resume execution if the statement evaluates to `false`. We then continue to iterate over the code in the current label, if any, adding additional `if-statements` to the table. We continue parsing any labels jumped to by recorded statements for additional `if-statements`, and subsequent labels. Once we reach a label without additional *if-statements*, we stop collecting. For the sample code in Figure 2, we extract the following grouping:

```
I0 (0) LABEL L1899798671
I4 (4) UNKNOWN Branch I4 IF_ICMPLE, jump to L1871531303
I5 (7) LABEL L717117575
I9 (10) UNKNOWN Branch I9 IF_ICMPGE, jump to L1871531303
I10 (13) LABEL L1866234461
I14 (17) UNKNOWN Branch I14 IF_ICMPLT, jump to L1871531303
I15 (20) LABEL L2064685037
I19 (25) UNKNOWN Branch I19 IF_ICMPEQ, jump to L1871531303
I20 (28) LABEL L1871531303
I23 (29) LABEL L2121337309
```

Next, we can connect the grouped statements through the order they are executed based on their evaluation: (1) We record the current label, where the expression resides. (2) We record the label that is jumped to if the expression evaluates to `true`. (3) We

record where execution resumes if the expression evaluates to `false`. This is either a continuation of the current label, or a new label that is reached immediately after the current expression. For the grouping above, we extract the following:

Expression	Location	True Jump Location	False Jump Location
I4	L1899798671	L1871531303	L717117575
I9	L717117575	L1871531303	L1866234461
I14	L1866234461	L1871531303	L2064685037
I19	L2064685037	L1871531303	L1871531303

This information indicates the order in which expressions are evaluated, and the outcome once they are evaluated. Using this information, we can form a truth table containing all possible paths through the gathered expressions. Each row of this truth table corresponds to a concrete test obligation that we impose for the Bytecode-MCC criterion. In order to achieve Bytecode-MCC, we need to cover all of the rows of the table. The truth table for the gathered expressions is shown to the right. From this table, the test obligations for the simple class in Figure 2 are: $(I4 = True)$, $(I4 = False \wedge I9 = True)$, $((I4 = False \wedge I9 = False) \wedge I14 = True)$, $(((I4 = False \wedge I9 = False) \wedge I14 = False) \wedge I19 = True)$, and $(((I4 = False \wedge I9 = False) \wedge I14 = False) \wedge I19 = False)$

I4	I9	I14	I19	Outcome Jump Location
True	-	-	-	L2089187484
False	True	-	-	L2089187484
False	False	True	-	L2089187484
False	False	False	True	L2089187484
False	False	False	False	L2089187484

3.2 Automated Test Generation to Satisfy Bytecode-MCC

Effective approaches to search-based generation require a fitness function that reports not just the percentage of goals covered, but *how close* the suite is to covering the remaining goals [15]. This feedback allows the search to efficiently maximize coverage of the chosen criterion.

In the case of Branch Coverage, the fitness function calculates the *branch distance* from the point where the execution path diverged from a targeted expression outcome. If an undesired outcome is reached, the function describes how "close" the targeted predicate was to the desired outcome. The fitness value of a test suite is measured by executing all of its tests while tracking the distances $d(b, Suite)$ for each branch.

$$F_{BC}(Suite) = \sum_{b \in B} v(d(b, Suite)) \tag{1}$$

Note that $v(...)$ is a normalization of the distance $d(b, Suite)$ between 0-1. The value of $d(b, Suite)$, then, is calculated as follows:

$$d(b, Suite) = \begin{cases} 0 & \text{if the branch is covered,} \\ v(d_{min}(b, Suite)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \tag{2}$$

The cost function used to attain the distance value follows a standard formulation based on the branch predicate [13]. Note that an expression must be executed at least twice, because we must cover the `true` and `false` outcomes of each expression.

In order to measure coverage of Bytecode-MCC and generate test cases intended to satisfy the produced obligations, we can make use of the same branch distance calculation. To obtain the fitness of a test suite, we calculate the branch distances for each expression (and desired outcome) involved in each obligation. Then, the fitness for an individual obligation is the sum of fitness values of all expressions (and desired outcomes) present in the obligation.

For each Boolean expression, we can calculate the minimal branch distance achieved by that suite. For each obligation, we calculate the branch distance for each targeted expression and outcome, then score that obligation as the sum of the branch distances for its targeted expression and outcome combinations. As execution comes closer to satisfying the obligation, the fitness should converge to zero. This fitness formulation can be used as a test generation target, or to measure coverage of existing test suites.

4 Study

We hypothesize that the simplified nature of bytecode instructions limits the effectiveness of tests by introducing the potential for masking, and that Bytecode-MCC-satisfying tests will be effective at overcoming this masking effect. Specifically, we wish to address the following research questions:

1. Does the Bytecode-MCC fitness function attain high coverage of the Bytecode-MCC test obligations?
2. Are test suites generated targeting Bytecode-MCC more effective at detecting faults than suites targeting Branch Coverage?
3. Does targeting the Bytecode-MCC and Branch Coverage fitness functions simultaneously yield higher levels of Bytecode-MCC coverage?
4. Does targeting the Bytecode-MCC and Branch Coverage fitness functions simultaneously yield higher levels of fault detection?

To address these questions, we have performed the following experiment:

1. **Collected Case Examples:** We have used 109 real faults, from five Java projects, as test generation targets.
2. **Generated Test Cases:** For each fault, we generated 10 suites targeting Bytecode-MCC, Branch Coverage, and a combination of both Bytecode-MCC and Branch Coverage for each class-under-test (CUT) using EvoSuite. We perform this process with both a two-minute and a ten-minute search budget per CUT.
3. **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed.
4. **Assessed Effectiveness:** For each fault and fitness target, we measure likelihood of fault detection (proportion of suites that detect the fault to the number generated).
5. **Measured Bytecode-MCC Coverage:** For each generated suite, we measure the attained Bytecode-MCC Coverage over the CUT.

Case Examples: Defects4J is a database of real faults extracted from Java projects [10]. The version used in this research, 1.20, consists of 395 faults from six projects:

Chart (26 faults), Closure (133 faults), Lang (65 faults), Math (106 faults), Time (27 faults), and Mockito (38 faults). As our focus is on complex Boolean expressions, we selected examples where the source code contains either a large number of Boolean expressions (at least 30), complex Boolean expressions (at least three conditions), or both. Following this filtering, we selected a subset of 109 faults: Chart (1), Closure (66), Lang (28), Math (11), and Time (4). For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the faults, and a list of classes and lines of code modified by the patch that fixes the fault.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The "fixed" version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactoring.

Test Suite Generation: We generate tests using EvoSuite targeting both Bytecode-MCC and Branch Coverage[1]. EvoSuite can also simultaneously target multiple criteria, with fitness evaluated as a single combined score. Therefore, we have also targeted a combination of Bytecode-MCC and Branch Coverage to evaluate whether the combination can achieve higher Bytecode-MCC coverage or detect more faults.

Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated using the fixed version of the CUT and applied to the faulty version because EvoSuite generates its own assertions for use as oracles. In practice, this translates to a regression testing scenario, where tests are generated using a version of the system understood to be "correct" in order to guard against future issues [17]. Tests that fail on the faulty version, then, detect behavioral differences between the two versions[2].

Two search budgets were used—two minutes and ten minutes per class. This allows us to examine whether an increased search budget benefits coverage or fault detection efficacy. These values are typical of other testing experiments [16]. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault, criterion, and search budget.

Generation tools may generate flaky (unstable) tests [17]. We automatically removed non-compiling test suites and tests that return inconsistent results over five trials. On average, less than one percent of the tests are removed from each suite.

5 Results and Discussion

The goal of our preliminary study is to determine whether search-based test generation is able to satisfy the test obligations of Bytecode-MCC within a typical search budget. We also wish to evaluate the fault-detection performance of the suites generated under that budget, regardless of the attained level of coverage.

[1] Specifically, the `onlybranch` fitness function, which omits branchless methods. This was chosen as our implementation of Bytecode-MCC also omits branchless methods.

[2] Note that this is identical practice to other studies using EvoSuite with Defects4J, i.e. [17,16]

5.1 Attained Bytecode-MCC Coverage

Table 1 lists the average Bytecode-MCC coverage attained given two-minute and ten-minute search budgets when targeting Bytecode-MCC alone and when targeting both Branch and Bytecode-MCC. From Table 1, we can see that the attained coverage is generally quite low. Overall, only 23.31% of obligations are covered on average under a two-minute budget, and only 25.53% under the ten-minute budget. On a per-system basis, the average ranges from 8.70% (Chart) - 69.58% (Time) under the two-minute budget and 15.40% (Chart) - 70.68% (Time) under the ten-minute budget.

	Two-Minute Budget		Ten-Minute Budget	
System	**MCC**	**MCC/BC**	**MCC**	**MCC/BC**
Overall	23.31	40.47	25.53	43.36
Chart	8.70	35.20	15.40	39.70
Closure	13.52	20.81	16.33	23.97
Lang	31.97	66.07	32.55	68.89
Math	41.00	67.04	44.22	69.22
Time	69.58	92.88	70.68	93.33

Table 1: Average Bytecode-MCC coverage (%) attained by test suites.

We can compare this to the attained Branch Coverage when targeting Branch Coverage as the optimization target, as detailed in Table 2. While these suites fail to attain 100% coverage of their targeted goal, these figures are much higher. Given two minutes for generation, these suites attain 71.39% more coverage of their stated goal (Branch Coverage) on average than suites targeting Bytecode-MCC (Table 1). Under a ten-minute budget, this increases to 86.72%.

System	Two-Minute Budget	Ten-Minute Budget
Overall	39.95	47.67
Chart	33.10	54.41
Closure	13.30	21.36
Lang	81.59	87.99
Math	73.00	77.36
Time	68.50	86.27

Table 2: Average Branch Coverage (%) attained by test suites targeting Branch Coverage.

What this shows is that Bytecode-MCC is a more difficult criterion to satisfy than Branch Coverage. Given the same period of time, we can naturally expect higher attainment of Branch Coverage than Bytecode-MCC coverage. Therefore, "typical" generation time frames like two minutes may not be enough to attain reasonable levels of Bytecode-MCC coverage. However, moving from two minutes to ten minutes offers only a 9.52% average improvement in attained Bytecode-MCC coverage, compared to an average improvement of 19.32% in Branch Coverage. The limited improvement suggests that an increased budget alone may not be enough to overcome the difficulty of satisfying Bytecode-MCC obligations.

This idea is further reinforced by examining the Bytecode-MCC coverage results when Branch Coverage and Bytecode-MCC are targeted simultaneously, as listed in Table 1 for each system and budget. Overall, targeting Branch and Bytecode-MCC coverage simultaneously yields a 73.62% increase in attained Bytecode-MCC coverage under a two-minute budget over targeting Bytecode-MCC on its own, and a 69.84% improvement under a ten-minute budget. Targeting Branch Coverage in addition to Bytecode-MCC offers easier-to-cover intermediate goals that, ultimately, result in improved Bytecode-MCC coverage. Coverage is still lower than desired, but the situation is improved over single-target optimization of Bytecode-MCC by introducing feedback (using Branch Coverage) that the test generator can work with.

5.2 Fault Detection

Table 3 lists the average likelihood of fault detection for suite generated to target Bytecode-MCC, Branch Coverage, and a combination of Bytecode-MCC and Branch

Coverage, divided by system and overall, for each search budget. Overall, Branch-targeting suites have a 21.20-22.13% likelihood of detection. This is consistent with previous experiments using this set of faults, and reflects the complex nature of the studied faults [16]. Overall, Bytecode-MCC-targeting tests only have a 4.27% average likelihood of detection under a two-minute budget, and a 3.47% average likelihood of detection under a ten-minute budget—far lower than when Branch Coverage is targeted.

This drop is likely due to the low coverage of Bytecode-MCC when it is the sole optimization target. Results improve when Bytecode-MCC and Branch Coverage are targeted simultaneously. Targeting both yields an overall average likelihood of detection of 16.67% (two-minute budget) and 19.33% (ten-minute budget). Still, this is lower than when Branch Coverage is targeted alone. Previous research indicates that multi-objective optimization can be more difficult than single-objective optimization [16], and it is likely that the additional burden of satisfying Bytecode-MCC results in lower Branch Coverage as well when both are targeted.

	Two-Minute Budget			**Ten-Minute Budget**		
System	**MCC**	**Branch**	**MCC/BC**	**MCC**	**Branch**	**MCC/BC**
Overall	4.27	21.20	16.67	3.47	22.13	19.33
Chart	20.00	100.00	90.00	1.00	100.00	90.00
Closure	0.00	1.33	1.00	0.00	3.67	2.33
Lang	8.28	41.72	28.28	7.24	43.79	32.41
Math	5.46	20.91	16.36	3.64	16.36	19.09
Time	0.00	2.50	32.50	0.00	0.00	35.00

Table 3: Average likelihood of fault detection (%) for each generation target and budget, broken down by system and overall.

However, if we examine results on a per-system basis, we can see that Bytecode-MCC satisfaction may have some promise for improvement in fault-detection. For the Time examples, targeting the combination of Branch Coverage and Bytecode-MCC yields over 92% Bytecode-MCC coverage on average. The combination also has an average likelihood of detection of 32.50-35.00%—well over the overall average. In this case, targeting the combination makes it possible to detect faults completely missed when targeting Branch Coverage alone.

On average, the Time examples contain more complex Boolean expressions than the other systems, with an average of 3.25 conditions per decision (compared to an overall average of 2.29). These are not trivial examples, and the performance when targeting the combination of Branch Coverage and Bytecode-MCC is promising. If systems contain complex Boolean expressions and high Bytecode-MCC coverage can be achieved, then we may also see improvements in fault detection. However, it is also clear that we must first find the means to improve attained Bytecode-MCC coverage.

5.3 Discussion

Even if Bytecode-MCC attainment is theoretically able to overcome issues with masking, we cannot test its abilities without first finding ways to improve coverage. The Time examples were the only ones where Bytecode-MCC coverage was reasonably high—particularly with the boost offered by simultaneously targeting Branch Coverage. While those showed promising improvements in fault detection as well, such improvements require increased ability to attain coverage.

Some criteria are inherently more difficult to satisfy than others [18]. It will be more difficult—and will require more test cases—to satisfy MCC over Branch Coverage. It may not be reasonable to expect equal coverage of Branch Coverage and Bytecode-MCC given the same time budget. Still, there may be means of improving coverage.

Reformulating the fitness function: A complicating factor in search-based test generation comes from the fitness function and its ability to offer feedback. When attempting to achieve Branch Coverage, the branch distance is used instead because it offers clear feedback, suggesting whether one solution is closer to covering the remaining obligations than another. It is possible that a fitness formulation other than the one employed in this work would yield better results. In the proposed function, each Bytecode-MCC obligation is a combination of smaller Boolean conditions. Fitness is measured by scoring each condition independently and linearly combining the resulting scores. Progress towards covering any of the individual conditions will yield a better fitness score. This, in theory, should offer reasonable feedback. However, there may exist cases where the independent subgoals conflict, and the choice of input may improve the coverage of one condition while increasing the distance for another condition.

A linear combination of condition distances may not be an ideal mechanism for judging fitness for Bytecode-MCC obligations, and other fitness formulations may yield better results. For example, it may be better to weight conditions based on the order they must be satisfied in. Alternatively, rather than combining the distances into a single score, each obligation could be treated as a set of distance scores to be optimized independently. This would be a more complex approach, but could potentially yield better results in cases where goals conflict.

Use Bytecode-MCC to measure adequacy instead of a direct generation target: Some criteria could yield powerful test cases, but lack sufficient feedback mechanisms to drive a search towards high levels of coverage. For example, Exception Coverage rewards test suites that throw many exceptions. However, there is no feedback mechanism that suggests "closeness" to throwing more exceptions [1]. This criterion yields poor suites when targeted as the sole fitness function, but offers great utility as a means of judging adequacy, and as a stopping criterion to determine when to finish testing. Likewise, we could target other fitness functions, but use Bytecode-MCC to assess the final test suites as a means to determine when to stop test generation.

Research has suggested that targeting unrelated fitness functions like Branch Coverage, or combining additional fitness functions with uninformative ones—i.e., Branch and Exception Coverage—results in higher Exception Coverage of the final suite [16]. Likewise, we could choose alternative optimization targets, then measure the attained Bytecode-MCC of the resulting tests. If we can find fitness functions that yield higher levels of Bytecode-MCC, we will be better able to evaluate the potential of the criterion for overcoming masking and improving the fault-detection potential of test suites.

Recent work explored the use of reinforcement learning to improve attainment of Exception Coverage [1]. The proposed approach was able to strategically adjust the targeted fitness functions over time in service of improving Exception Coverage. A similar adaptive fitness function selection approach could be used to discover combinations of fitness functions that attain high coverage of Bytecode-MCC.

We may wish to also consider other forms of test generation, beyond search-based generation. For example, (dynamic) symbolic execution techniques use sophisticated solvers to attain input designed to drive program execution towards particular paths [2]. Such approaches suffer from limitations in terms of the type of programs and language features they can handle, and in terms of scalability [5]. However, they can be very

effective at producing the input needed to traverse specific paths—which is required for Bytecode-MCC satisfaction. The use of symbolic execution—or approaches that combine search and symbolic execution—may be required to achieve high levels of Bytecode-MCC coverage.

6 Threats to Validity

Internal Validity: Because EvoSuite's test generation process is non-deterministic, we have generated ten test suites for each combination of fault, budget, and fitness function. It is possible that larger sample sizes may yield different results. However, we believe that this is a sufficient number to draw stable conclusions.

External Validity: Our study has focused on only five systems. We believe that such systems are representative of, at minimum, other small to medium-sized open-source Java systems. We believe that we have chosen enough examples to gain a basic understanding of Bytecode-MCC, and that our results are generalizable to sufficiently similar projects. In this study, we have implemented Bytecode-MCC within EvoSuite. Results may differ using a different test generation algorithm. However, we believe that EvoSuite is sufficiently powerful to explore our proposed ideas.

7 Related Work

Hayhurst et al. observed sensitivity to statement structure, stating that "if a complex decision statement is decomposed into a set of less complex (but logically equivalent) decision statements, providing MC/DC for the parts is not always equivalent to providing MC/DC for the whole" [11]. Gargantini et al. have also observed the sensitivity of structural coverage metrics to modification of the code structure and proposed a method of automatically measuring the resilience of a piece of code to modification [6]. Chilenski further made the observation that "If the number of tests M is fixed at N + 1 (N being the number of conditions), the probability of distinguishing between incorrect functions grows exponentially with N, $N > 3$" [3]. This observation is based on the number of tests, but notes that testing power grows with statement complexity.

In past work, we empirically demonstrated the effects of expression structure on coverage and suite effectiveness, clearly illustrating the negative impact of statement simplification on a series of industrial case examples [8]. Our results supported the previous observations. We also proposed a set of "observability" extensions for source-based coverage criteria to overcome masking between-expressions [14,18]. Our proposed method, Bytecode-MCC acts in a similar—but more limited—manner to the notion of "observability", requiring that masking be overcome in closely-connected statements. To date, ours is the first approach to address masking in search-based test generation or when considering bytecode representations of programs.

8 Conclusions

Masking occurs when one condition prevents another from influencing the output of a Boolean expression. Adequacy criteria such as Multiple Condition Coverage (MCC) overcome masking within one expression, but offers no guarantees about subsequent

expressions. As a result, a Boolean expression written as a single complex statement will yield more effective test cases than when written as a series of simple expressions. Many approaches to automated test case generation for Java operate not on the source code, but on bytecode. The transformation to bytecode simplifies complex expressions into multiple expressions, introducing masking. We propose Bytecode-MCC, a new adequacy criterion designed to group bytecode expressions and reformulate them into complex expressions. Bytecode-MCC should produce test obligations that are more likely to reveal faults in program logic than tests covering the simplified bytecode.

A preliminary study conducted over 109 faults from Defects4J indicate the potential of the technique. Results attained for the "Time" system, where targeting the combination of Bytecode-MCC and Branch Coverage yields high Bytecode-MCC coverage, show fault detection well above the overall average. However, there are multiple research challenges to be overcome. Bytecode-MCC is more difficult to achieve than Branch Coverage, and its fitness function does not offer sufficient feedback to guide test generation. This suggests that Bytecode-MCC may be best used as a method of judging test suite quality, rather than as a direct generation target. Simultaneously targeting Bytecode-MCC and Branch Coverage improves coverage of Bytecode-MCC and the likelihood of fault detection. It may be possible to identify other fitness functions that are effective at attaining Bytecode-MCC.

In future work, we will explore methods of improving Bytecode-MCC coverage. In particular, we plan to: (1) Explore alternative formulations of the fitness function for Bytecode-MCC, such as applying weights based on the order that sub-obligations must be solved. (2) Examine the use of Bytecode-MCC as a way to judge test suites generated targeting other criteria, as well as its use as a stopping condition for test generation. (3) Investigate the use of reinforcement learning to automatically identify alternative generation targets that will yield higher attainment of Bytecode-MCC than direct targeting of Bytecode-MCC during test generation. (4) Vary the algorithms used to generate Bytecode-MCC-covering test suites.

References

1. Almulla, H., Gay, G.: Learning how to search: Generating exception-triggering tests through adaptive fitness function selection. In: 13th IEEE International Conference on Software Testing, Validation and Verification (2020)
2. Anand, S., Burke, E., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey on automated software test case generation. Journal of Systems and Software 86(8), 1978–2001 (August 2013)
3. Chilenski, J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Tech. Rep. DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C. (April 2001)
4. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 291–301. ISSTA, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2483760.2483774
5. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 360–369 (Nov 2013)

6. Gargantini, A., Guarnieri, M., Magri, E.: Aurora: Automatic robustness coverage analysis tool. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. pp. 463–470. IEEE (2013)
7. Gay, G., Staats, M., Whalen, M., Heimdahl, M.: The risks of coverage-directed test case generation. Software Engineering, IEEE Transactions on PP(99) (2015)
8. Gay, G., Rajan, A., Staats, M., Whalen, M., Heimdahl, M.P.E.: The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. ACM Trans. Softw. Eng. Methodol. 25(3), 25:1–25:34 (Jul 2016), http://doi.acm.org/10.1145/2934672
9. Groce, A., Alipour, M.A., Gopinath, R.: Coverage and its discontents. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. pp. 255–268. Onward!'14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2661136.2661157
10. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 437–440. ISSTA 2014, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2610384.2628055
11. Kelly J., H., Dan S., V., John J., C., Leanna K., R.: A practical tutorial on modified condition/decision coverage. Tech. rep. (2001)
12. Li, N., Meng, X., Offutt, J., Deng, L.: Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 380–389 (Nov 2013)
13. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14, 105–156 (2004)
14. Meng, Y., Gay, G., Whalen, M.: Ensuring the observability of structural test obligations. IEEE Transactions on Software Engineering pp. 1–1 (2018), available at http://greggay.com/pdf/18omcdc.pdf
15. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: Barros, M., Labiche, Y. (eds.) Search-Based Software Engineering, Lecture Notes in Computer Science, vol. 9275, pp. 93–108. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-22183-0_7
16. Salahirad, A., Almulla, H., Gay, G.: Choosing the fitness function for the job: Automated generation of test suites that detect real faults. Software Testing, Verification and Reliability 29(4-5), e1701 (2019), https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1701, e1701 stvr.1701
17. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). ASE 2015, ACM, New York, NY, USA (2015)
18. Whalen, M., Gay, G., You, D., Heimdahl, M., Staats, M.: Observable modified condition/decision coverage. In: Proceedings of the 2013 Int'l Conf. on Software Engineering. ACM (May 2013)
19. Whalen, M.W., Person, S., Rungta, N., Staats, M., Grijincu, D.: A flexible and non-intrusive approach for computing complex structural coverage metrics. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. pp. 506–516. IEEE Press (2015)